CISCO SYSTEMS

Document Number	EDCS-201202
Revision	0.4
Author	Carol Gal, Doug Wooff
Project Manager	Mala Devlin

Project Name: HFR

HFR Boot Architecture Functional Specification

Project Headline

This document describes the HFR system boot architecture and applicable scenarios. Areas of performance improvements are identified and solutions proposed in the second part of the document.

Reviewers

Department	Name and Title
Project Manager	Mala Devlin, Manager, DSM Software
Engineering	David Ward, HFR Software Architect Ayman Sayed, Director, HFR Software Development
Product Marketing	Ashok Ganesan
Test	Richard Mackay, Manager Devtest Manishee Gupta, Manager Devtest
Quality	???

Modification History

Rev	Date	Originator	Comment
0.1	[redacted]	Carol Gal	First pre-draft version
0.2	[redacted]	Carol Gal	Took out packaging/MBI upgrade, added figure
0.3	[redacted]	Doug Wooff	Add QFT3 scope, updates
0.4	[redacted]	Doug Wooff	Update to FCS status

EXHIBIT 1.2

1.0 Introduction

This document provides an overview of the HFR boot architecture for FCS and the distributed functionality of various components involved in the boot process. The scope of this document is limited to the FCS development time frame. As such, Logical Routers (LR's) are not fully supported by Install components. In this respect, there is only one LR for the entire multi-rack system for FCS. LR's are still mentioned in this document for future consideration, but the reader should bear in mind that only one (implicit) LR will be supported for FCS, and therefore whatever LRd usage is discussed herein is the best projection currently available.

Different boot scenarios are described from a timeflow perspective. Areas of performance improvements are identified and solutions proposed - their feasibility and staged implementation to be determined.

1.1 Purpose and Problem Definition

The HFR system is a distributed, multi-chassis highly scalable router [1,2]. With the size of such a large system, particular problems have to be solved with respect to the boot time performance as well as software modularity. Boot time performance and determinism is addressed by having a hierarchical approach to the boot process, in which concurrency and parallelism are applied as we move from top to bottom. Software modularity implies being able to upgrade or downgrade software packages without impacting the main function of the system, routing and forwarding packets. Both of these requirements are major factors in achieving the high availability target of "five nines" for HFR.

The FRD document [8] introduces the requirements for boot time and package upgrade / downgrade, while this document describes the boot architecture and different scenarios.

1.2 Functions

For FCS development, multiple Logical Routers (LR's) will not be supported. All racks and cards are assumed to belong to the one and only (implicit) LR. This does not affect the design of the boot architecture, however. All booting nodes are served by an authority one level higher in the validation chain, with or without LR support.

1.3 Audience

This document is intended for the design and testing HFR community or anyone else who wants to know the mechanics of the boot process. Familiarity with HFR is a must [9,10], while the other documents in the reference section provide detail descriptions of various components involved in the boot process.

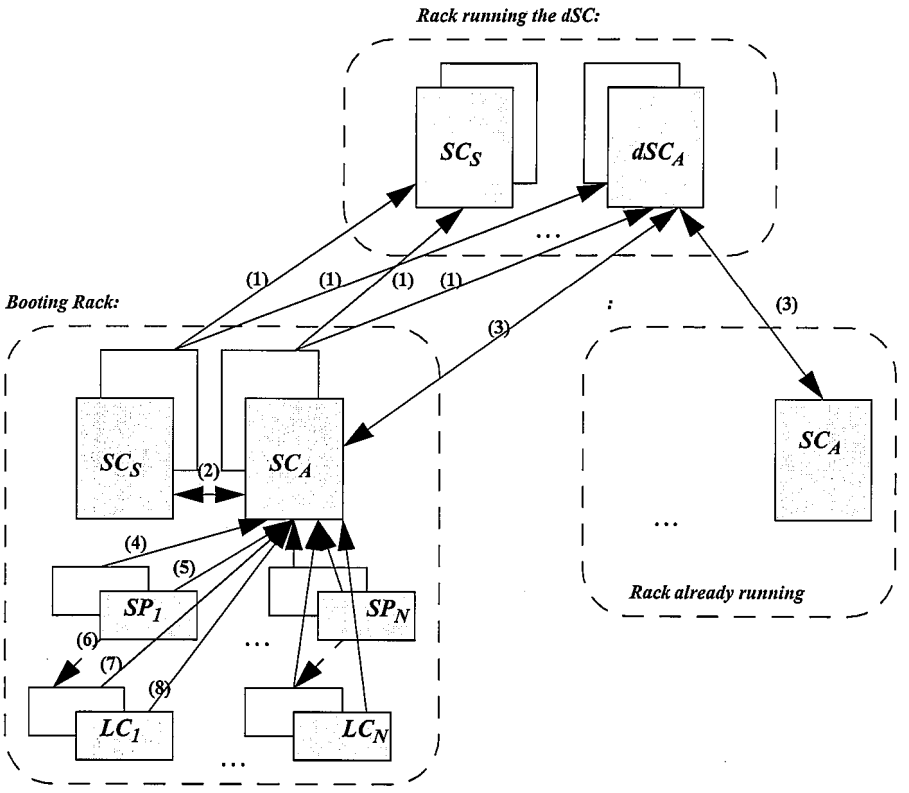
2.0 Functional Overview

2.1 System Overview

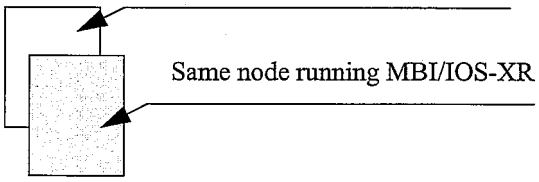
The validation "chain of command" can be described quite simply. All SP/LC/DRP cards are served by the local active System Controller (SC). Each active SC is in turn served either by the only one dLRSC (using LR's), or by the dSC when LR's are not enabled. Note that the System Controller is sometimes called the Shelf Controller, and that the RP on line card racks takes on the role of System Controller.

In an LR-enabled system, dLRSC's would be served by the dSC. The dSC would not need to directly validate any other SC nodes in the LR-enabled system.

It should be clear that we need only replace 'dSC' in this single-LR discussion with 'dLRSC' of future multi-LR systems for boot architecture equivalency.



LEGEND:



The node can be:

-$SC_{A(S)}$: Shelf Controller Active (Standby)
-dSC: Designated Shelf Controller
-SP: Service Processor
-LC: Line Card

The boot process of a rack is an orderly, hierarchical process with well defined "boot segments" that do not overlap in time. Parallelism is achieved within these boot segments by booting many similar nodes at the same time, but the duration of each individual boot segment is an additive factor for the overall rack boot time. During each boot segment we identify separately the ***booting node*** from the node that is validating the node's boot process, the ***boot server node***. These segments are as follows, with reference to the numbers in figure X above.

I. SC boot (booting node SC - boot server node dSC):

(1) - Each SC ROMMON in the shelf independently configures its Ethernet switch and tries to contact the dSC with BOOT_REQUEST messages via the GigE port. The dSC ShelfMgr replies, and as a result, an MBI is launched. Then, by one of two methods (see Note1), IOS-XR starts running on each SC node.

(2) - The two SCs arbitrate between themselves for the Active/Standby role.

(3) - The active SC reconfigures its Ethernet switch and becomes the boot server node for the rack. From now on, BOOT_REQUEST messages from SP/LC/DRP cards in the rack are directed at the active SC. In addition, the active SC monitors the beacon from the dSC to learn of any change of *its* boot server node.

Note1: A node can boot an install stream either by discovering it in local storage and authorizing it, or by having the launched MBI pull packages down from the boot server node. Similarly, the launched MBI itself can either be discovered locally and authorized, or a new MBI can be pulled down via tftp by ROMMON. An 'install stream' is a compatible set of packages plus an associated MBI.

II. SP boot (booting node = SP, boot server node = SC_A):

(4) - Each SP ROMMON in the shelf contacts the SC_A with BOOT_REQUEST messages. The SC_A ShelfMgr replies, and as a result, a validated MBI is launched.

(5) - Each SP Insthelper validates its local installed software with SC_A. If needed, packages are pulled from SC_A and launched, otherwise locally cached packages are used. IOS-XR starts running on the SP node.

(6) - ShelfMgr on SC_A instructs the SP to power on the LC/DRP attached to it (for cards that are not SP-only).

III. LC/DRP boot (booting node = LC/DRP, boot server node = SC_A):

(7) - Each powered-on LC/DRP ROMMON in the shelf contacts the SC_A with BOOT_REQUEST messages. The SC_A ShelfMgr replies, and as a result, a validated MBI is launched.

(8) - Each LC/DRP Insthelper validates its local installed software with SC_A. If needed, packages are pulled from SC_A and launched, otherwise locally cached packages are used. IOS-XR starts running on the LC/DRP node. The LC/DRP is now fully booted.

2.2 Feature Listing

3.0 External Specifications (System Requirements)

Please see EDCS-194266 for a summary of the system requirements.

3.1 Overview

3.2 Functional Requirements

3.3 Performance Requirements

3.4 External Interface Requirements

3.5 Design Constraints

3.6 Quality Requirements

3.7 Security Considerations

4.0 Boot Scenarios and Dependencies

The key software components involved in booting / upgrading an HFR node are:

- ROMMON
- MBI
- MBI Manager
- Distributed Software Management: Install Director (instdir), formerly known as Install Manager (Instmgr), Install Helper (Insthelper)
- Plugin Controller (PCtl), aka 'Shelf Manager' (ShelfMgr), and Designated Shelf Controller Process (dSCp)
- Logical Router Daemon (LRd)

Please consult the reference documents for a detailed description of each component and its role in HFR booting and upgrade scenarios. The following paragraphs describe different usage scenarios and the overall flow and interaction between these key components - as they are currently envisioned to be delivered for FCS. Based on these scenarios, the Appendix then identifies areas of future performance improvements and details the required changes and the key components affected.

4.1 Cold Rack Boot

In this section, 'cold boot' of racks is described. Here, 'cold boot' means the first bringup of an HFR system, comprising at least one rack and at most one Logical Router. It is assumed that the a user with owner privileges will configure an LR boundary and add to the LR inventory all the desired cards (DRPs, LCs). Every rack will have a pair of SC/SCRP cards (active / hot standby) that will be shipped with some factory or custom software packages programmed into their internal flash storage device. DRPs will be inserted into any LC slots and can also have default packages pre-installed in their flash disk devices. DRP's *should* be installed in pairs for redundancy purpose. Line Cards (LCs) do not have flash disk devices, but instead will use bootflash for MBI and package storage.

4.1.1 Single Rack HFR System

In this simplest example, we assume that the only rack in the system is the one that is being booted and it is configured as a single Logical Router. The sequence of events is presented below and can be logically grouped, and separately described, as booting A) the SC's B) the SP's and C) LC/DRP nodes.

A. SC node boot:

1) Upon rack power on, the SC and SP cards are the only cards that are automatically powered on by their hardware; they then execute in ROMMON.
2) The two SC ROMMON's will each send a BOOT_REQUEST message over the control backplane GE to a well known MAC address trying to contact the dSC. This message attempts to validate their rack number plus the locally cached MBI versions. These boot request messages are sent at regular intervals until a response is received *or* there is a timeout (10 seconds).
3) As there is no dSC in the HFR system to reply, the above request will timeout and a default locally cached MBI is chosen to load and execute. If none is available, the SC will stay in ROMMON and not proceed further, except to continue sending BOOT_REQUEST messages, and scanning for newly available MBI's (from an inserted flash disk, for example).
4) Within the SC MBI, install software discovers an existing LOADPATH from factory-installed packages. These packages are normally installed to the internal disk, and LOADPATH then points to packages on the same disk. System Manager starts spawning the server processes in the packages.
5) Processes that are responsible for A/S arbitration are started (reddrv and redcon) and the role of the SC determined: active or standby. This information is relayed to Sysmgr that will continue spawning the server processes accordingly: some processes are only started when the SC is active and others might be in standby mode.

6) Once Sysmgr has finished spawning all server processes, the SCs are completely booted and there is an active and a standby SC in the rack. The active SC becomes the boot server source for the entire system (dSC), since there are no other contenders, as well as the boot server source for the rack (is the local SC). The Shelf Manager on the active SC is responsible for managing all cards present within the rack.

7) If the standby SC, which booted independently and in parallel with the active SC (also the dSC) doesn't have the proper MBI or software stream, it will be instructed to reset by the instdir on dSC and directed to get the proper versions upon its next boot.

B. SP node boot:

8) Each SP card in the rack has launched its ROMMON upon power on. It then sends a (rack local) broadcast BOOT_REQUEST message at regular intervals trying to contact the $Instmgr_{SC}$ (the boot server source for the rack). The BOOT_REQUEST message contains the card type, slot/instance number, and a list of locally cached MBIs (if any) for the booting node. Until a reply is received, the SP ROMMON will not proceed further (there is no timeout like in the SC case). The SC, by having properly initializing the Broadcom switch, will ensure that all these messages generated from within the rack do not cross rack boundaries and therefore the slot/instance numbers uniquely identify the sender.

9) $Instmgr_{SC}$ is handling these requests that are received via ShelfMgr. After consulting with LRd about the membership of the card, ShelfMgr (using data from mbimgr) replies with a unicast message to each of these BOOT_REQUEST messages, filling in the rack number and the selected MBI. If the BOOT_REQUEST message did not reference the correct (locally cached) MBI, Shelfmgr replies with the proper MBI version and the tftp server IP address from which to obtain it.

10) ROMMON SP receives the reply and properly configures its MAC address. If the selected MBI is locally cached then it is immediately loaded and launched. If not, then it is downloaded from the tftp server (using tftp services over Ethernet).

11) The SP MBI is launched and Insthelper process spawned. The MBI-Hello process comes to life, and contacts ShelfMgr. ShelfMgr sends a "BSS" message back to MBI-Hello indicating the boot server source. Insthelper reads the boot server source node from MBI-Hello and contacts $Instmgr_{SC,}$ using LWM and QNet services (only), requesting the LOADPATH appropriate for the node. Insthelper constructs a well-known QNet path to locate the Instmgr on the boot server, since QSM symlinks are not available at this time in the boot sequence.

12) If the selected MBI is not local, a copy is cached locally on the persistent storage device for future use. Once the MBI is cached (ie. burned to bootflash), a self-reset is initiated to actually boot from the new MBI.

13) $Instmgr_{SC,}$ contacts the LRd for the membership of the requesting SP node and scans the software configuration data to find the install stream that matches the card type / RSI address of the node. $Instmgr_{SC}$ replies to each node's request with the LOADPATH.

14) Insthelper installs the software by downloading packages over QNet, writing the packages to local storage (flash or memory, as appropriate), then updates the LOADPATH and passes it to Sysmgr, who starts spawning the processes. The SP becomes fully functional, running IOS-XR.

15) Shelf Manager on the SC validates the LC/DRP card type that the SP is attached to (ie. for which the SP is a daughter board) and its configured state. If approriate, the mother card (ie. LC/DRP) is powered on, otherwise kept in reset.

C. LC/DRP node boot

16) Steps 7-13 (above) are followed indentically by an LC or DRP node. The only difference is that LC's and DRP's are not powered on by default, but under control of ShelfMgr.

4.1.2 Multi-rack HFR System

In this example we assume that the HFR system being powered on consists of multiple racks. The main difference from the single rack system is the fact that the booting SCs will have to sort out their role in the system: either they will find an existing dSC or will have to elect one through an election algorithm involving communications over the control backplane GE. The election protocol kicks in at the stage when the SCs are running the full IOS-XR image and continues forever, even after a dSC was elected, thus permitting a re-election process triggered dynamically, without the need to bring down an entire rack. For details, please consult ENG-155856 and ENG-114800.

Once a dSC has been elected and the install stream distributed to all SCs in the system, booting the nodes within the rack is identical to the steps described before for SP and LC/DRP nodes: their boot server node is always the active SC in their rack. Therefore, only the SC node boot will be presented bellow.

4.1.2.1 Rack OIR

We assume that there is already a running HFR system with an elected dSC running in one of the racks. We power on a new rack within this system.

1) Power is applied to the new rack. Only the SC cards and the SP cards are powered on. SP cards start sending BOOT_REQUEST messages (confined to the boundaries of their rack) that will not be replied to until one SC becomes active, and the relevant processes (ShelfMgr, Instmgr) start up, initialize, and synchronize themselves with their surroundings. When an SC becomes active, it is running an IOS-XR set of packages (ie. install stream), the rack boot server source is resolved, and Instmgr knows which packages are configured for the local nodes.
2) The two SC ROMMON will each send a BOOT_REQUEST message over the control backplane GE to a well known MAC address trying to contact the dSC in order to validate their rack number and the locally cached MBI version to be launched. These boot request messages are sent at regular intervals until a response is received *or* there is a timeout
3) The Shelf Manager running on dSC receives these BOOT_REQUEST messages, obtains a rack number for the newly inserted rack and replies with a selected MBI (either locally or from a tftp server). The rack number is assigned as a configuration against the rack serial number. *Note: In absence of a matching configuration, the BOOT_REQUEST message is ignored. The absence of an appropriate configuration could cause a newly booting SC to time out and boot whatever local software may have been previously installed (eg. at the factory). In addition, the booting SC may assume that it is (also!) rack number zero (which is the default for the dSC rack), and cause application collisions when it boots. Users need to understand this and avoid this scenario when first booting a second, third, fourth rack.*
4) SC ROMMON receives the reply and proceeds to load either the local MBI from disk or it downloads the correct remote one via tftp over ethernet.
5) The SC software is launched, starting with only the MBI. insthelper within the newly launched MBI contacts instdir on dSC to obtain the appropriate LOADPATH. A sync is done to ensure that all correct packages are available on its local disk, then the packages for the new SC itself are launched, and it starts running IOS-XR.
6) Processes that are responsible for A/S arbitration are started first (reddrv and redcon) and the role of the SC determined: active or standby. This information is relayed to Sysmgr that will continue spawning the server processes accordingly: some processes are only started when the SC is active and others might be in standby mode
7) Once Sysmgr has finished spawning all server processes, the SCs are completely booted and there is an active and a standby SC in the rack. The active SC becomes the boot server source for the rack. The Shelf Manager on the active SC is responsible for managing all cards present within the rack. From now on, the SC will send at regular intervals a PRESENCE message to the dSC and will receive a BEACON message from the dSC. These ethernet messages track the OIR status of each rack in the system, as well as conveying the dSC health. There is no inter-rack hardware control signal in the current design.

8) SP nodes have been sending BOOT_REQUEST messages all this time. Only when Shelfmgr is up and running are the messages acknowledged. The SP's are then booted, followed by the LC/ DRP nodes - see single rack HFR system description

4.1.2.2 Multi-rack system cold boot

We assume that the whole HFR system is booted at the same time and it is comprised of two or more racks.

1) Power is applied to all racks at the same time, there is no dSC running. Only the SC cards and SP cards are powered on. SP cards start sending BOOT_REQUEST messages (confined to the boundaries of their rack) that will not be replied to until a local SC becomes active.
2) All SCs ROMMON will each send a BOOT_REQUEST message over the control backplane GE to a well known MAC address trying to contact the dSC in order to validate their rack number and the locally cached MBI version to be launched. These boot request messages are sent at regular intervals until a response is received *or* there is a timeout
3) As there is yet no dSC elected in the HFR system to reply, the above requests will time-out and a default locally cached MBI is chosen to load and execute - on each SC.
4) The SC software is launched and IOS-XR is running. Instmgr reads the local software configuration (which is assumed to be correct) to find the install stream and sets up the LOADPATH. System Manager starts spawning the server processes, including the dSC process (dSCp) that is responsible for the dSC election
5) The SC arbitration (A/S) within each rack occurs and Sysmgr spawns all the processes.
6) Each dSCp will wait for a BEACON message from the elected dSC and will timeout as none exists. It will then broadcast QUERY messages containing its serial number, rack number and user assigned priority in a bid to become dSC. In the meanwhile, it will process all QUERY messages it receives from the other SCs.
7) Upon receiving a better QUERY message than its own, the dSCp changes state and waits for the BEACON message. If no better QUERY message is received, that particular SC assumes the status of the dSC and starts broadcasting BEACON messages. From now on, all the SCs in the system will take note of the address of the elected dSC contained in the periodic BEACON message and will send periodically a PRESENCE message of their own.
8) Once a dSC has been elected, and instdir is running there, all other insthelper instances on other SC's, both active and standby, validate their own locally booted install stream and MBI. Incorrect packages or an invalid MBI on the newly booted SC will be copied locally and launched. A serious mismatch will lead to the SC being instructed to reset itself, and start the boot process anew.

Notes:

- *on cold boot, only the SCs will come up first, electing a dSC*
- *any new install software will have to be distributed to the other SCs*
- *the other cards in the rack boot next, first the SP (ROMMON and image), then LC/DRP (ROMMON, MBI, packages). These steps are on a one-to-one basis between the cards and their respective SC*
- *cards could be added also after boot one by one, by inserting, registering them with the LR and boot them*

4.2 Warm Rack Boot

A "warm" rack boot refers to a rack boot when all of the code for all nodes has been previously cached in persistent storage. It is expected that a warm boot will proceed quickly, and will restore the system to its previous state in a deterministic manner. Any configurations would have been previously entered and committed, such that they will be applied as the rebooted rack(s) reach steady state.

It is acknowledged that warm reboots of racks should normally not be needed during the life of an HFR system. Exceptional circumstances may arise, such as a power outage, where it is unavoidable. HFR must rapidly reboot and resume its previous functioning. This is the situation when a stringent boot time requirement is in place, as described in R-3.2.1 in the FRD document [8].

4.2.1 Single Rack Warm Boot

The event flow is very similar to the cold boot case.

After a power cycle, both SC's in the rack boot their rommons, since they have power automatically applied. They both send out BOOT_REQUEST messages, trying to contact the dSC. There is no dSC running, so they both timeout having received no response, and launch the local software on their disks.

They will both soon be running IOS-XR, and one of them will be elected as dSC. instdir will run on the dSC, and the insthelper on the non-dSC will validate its LOADPATH against the one on dSC. If they were different, it would self-reset, so that all of its software would get aligned (ie. sync'd) to the dSC software. Since the scenario here is one where both SC's were previously installed with the same install stream, the LOADPATH will match, and the non-dSC will continue running.

All SP's have been automatically powered on and have been sending BOOT_REQUEST messages - with no answer. As soon as IOS-XR is running on the primary SC in the rack, Shelfmgr/Mbimgr can validate the SP's cached MBI's and packages for quick launch.

Following soon after, all LC/DRPs present in the rack will be powered on and boot in parallel. Both the MBI and install stream is already cached locally for each node. Each MBI and package set will receive a quick validation, and will be immediately launched without the need for making fresh local copies.

4.2.2 Multiple Rack Warm Boot

After a power cycle of several racks, all SC's in all racks boot their rommons, since they have power automatically applied. They all send out broadcasted BOOT_REQUEST messages, trying to contact the dSC. There is no dSC running, so they all timeout having received no response, and launch the local software on their disks. At this time, the rack number is fetched from the backplane IDEPROM, and used to boot the software. The rack number had been burned into theIDEPROM during initial cold boot.

All SC's will soon be running IOS-XR, and one of them will be elected as dSC. instdir will run on the dSC, and the insthelpers on the non-dSC SC nodes will validate their LOADPATH against the one on dSC. If they were different, they would self-reset, so that all of its software would get aligned (ie. sync'd) to the dSC software. Since the scenario here is one where all SC's were previously installed with the same install stream, the LOADPATH will match, and the non-dSC SC's will continue running.

The warm boot scenario for a single rack is then played out within each rack. All SP's have been automatically powered on and have been sending BOOT_REQUEST messages - with no answer. As soon as IOS-XR is running on the primary SC in the rack, Shelfmgr/Mbimgr can validate the SP's cached MBI's and packages for quick launch.

Following soon after, all LC/DRPs present in the rack will be powered on and boot in parallel. Both the MBI and install stream is already cached locally for each node. Each MBI and package set will receive a quick validation, and will be immediately launched without the need for making fresh local copies.

Appendix A References

1. HFR DSM Boot Services, EDSC-131905
2. HFR Distributed Software Management, ENG-150583
3. HFR DSM QFT2 Functional and Design Specification, ENG- 126996
4. HFR MBI Manager Functional Specification, ENG-136393
5. HFR LR Daemon Functional Specification, ENG-108732
6. HFR Designated Shelf Controller dSC Functional Specification, ENG-114800
7. HFR Designated Shelf Controller Design Specification, ENG-155856
8. HFR Boot Architecture and Optimization FRD, ENG-194266
9. HFR HW System Functional Specification, ENG-78989
10. HFR Distributed Systems Architecture Overview, ENG-75484